Struktura projektu obiektowego

1. Podejście proceduralne – zadania podzielone się na podprogramy w postaci procedur i funkcji, a główny blok programu zawiera kolejność wykonywanych operacji.

```cpp
#include <iostream>

using namespace std;

double convertToK(double temperature)
{
    double constant = 273.15;
    cout << "Changing units of temperature from C to K \n";
    return temperature + constant;
}

double convertToF(double temperature)
{
    double factor = 1.8;
    int constant = 32;
    cout << "Changing units of temperature from C to F \n";
    return factor * temperature + constant;
}

int main()
{
    double tempC;
    cout << "Insert temperature in C \n";
    cin >> tempC;
    double tempK = convertToK(tempC);
    double tempF = convertToF(tempC);
    cout << "Temperature in K is equal to: " << tempK << endl;
    cout << "Temperature in F is equal to: " << tempF << endl;
    return 0;
}
```

nagłówek funkcji (header)

ciało funkcji (source)

```cpp
#include <iostream>

using namespace std;

double convertToK(double temperature);

double convertToF(double temperature);

int main()
{
    double tempC;
    cout << "Insert temperature in C \n";
    cin >> tempC;
    double tempK = convertToK(tempC);
    double tempF = convertToF(tempC);
    cout << "Temperature in K is equal to: " << tempK << endl;
    cout << "Temperature in F is equal to: " << tempF << endl;
    return 0;
}

double convertToK(double temperature)
{
    double constant = 273.15;
    cout << "Changing units of temperature from C to K \n";
    return temperature + constant;
}

double convertToF(double temperature)
{
    double factor = 1.8;
    int constant = 32;
    cout << "Changing units of temperature from C to F \n";
    return factor * temperature + constant;
}
```

nagłówek funkcji (header) + średnik

ciało funkcji (source)

Powyżej przedstawione są 2 przykłady kodu napisanego w sposób proceduralny. Gdyby funkcje przedstawione w pierwszym przykładzie zawierały dużą liczbę linijek lub gdyby samych funkcji było dużo więcej, trzeba byłoby przewijać kod żeby dostać się do funkcji main(). Stąd, wygodniejszym sposobem jest zadeklarowanie używanych funkcji poprzez wpisanie jedynie ich nagłówków przed main(), a następnie opisanie ciał funkcji.

2. W dużych projektach obiektowych nagłówki funkcji znajdują się w osobnym pliku z rozszerzeniem *.hpp (pliki nagłówkowe), a ich ciała w plikach *.cpp (pliki źródłowe). W osobnym pliku *main.cpp* zamieszcza się główną funkcję zarządzającą main().

header.hpp

```
#include <iostream>

using namespace std;

double convertToK(double temperature);

double convertToF(double temperature);
```

main.cpp

```
#include <iostream>
#include "headers.hpp"

using namespace std;

int main()
{
    double tempC;
    cout << "Insert temperature in C \n";
    cin >> tempC;
    double tempK = convertToK(tempC);
    double tempF = convertToF(tempC);
    cout << "Temperature in K is equal to: " << tempK << endl;
    cout << "Temperature in F is equal to: " << tempF << endl;
    return 0;
}
```

sources.cpp

```
#include <iostream>
#include "headers.hpp"

using namespace std;

double convertToK(double temperature)
{
    double constant = 273.15;
    cout << "Changing units of temperature from C to K \n";
    return temperature + constant;
}

double convertToF(double temperature)
{
    double factor = 1.8;
    int constant = 32;
    cout << "Changing units of temperature from C to F \n";
    return factor * temperature + constant;
}
```

Kompilacja:
g++ *main.cpp header.hpp sources.cpp*

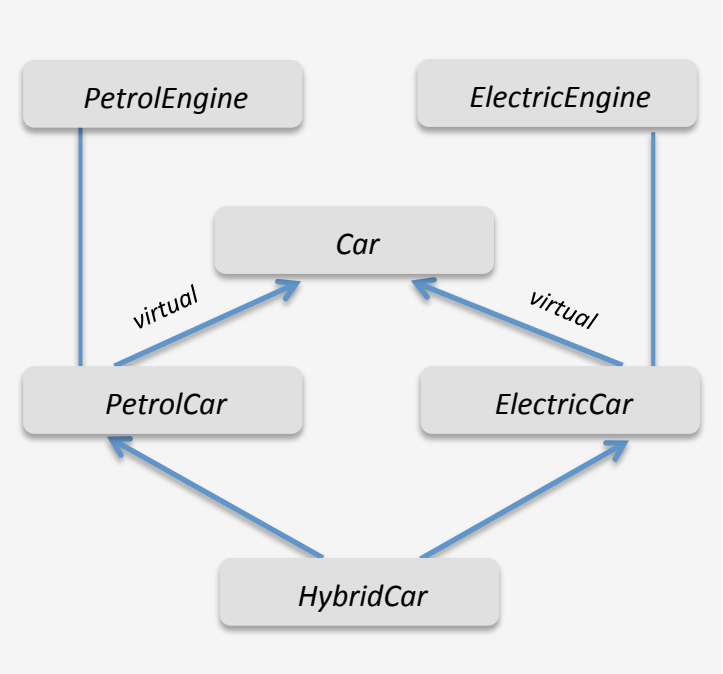
W dużym projekcie obiektowym, ze względu na dużą liczbę plików źródłowych, kompilację, tj. budowanie programu przeprowadza się za pomocą programów *cmake* i *make*. W tym celu tworzy się katalog *build* oraz plik wejściowy do programu *cmake*, tj. *CMakelist.txt*. Budowanie programu tworzy się za pomocą poleceń: 1) *cd build* − wejście do katalogu 2) *cmake ..* − wywołanie programu cmake do tworzenia pliku *Makefile* na podstawie *CMakeLists.txt* 3) *make* − kompilacja programu. W efekcie otrzymujemy plik wykonywalny o nazwie projektu. Uruchomienie poleceniem *./nazwa_projektu.*

3. Struktura projektu obiektowego na przykładzie repozytorium *coders-school/Cars*

Drzewo katalogu *Cars*

```
.
├── CMakeLists.txt
├── Car.cpp
├── Car.hpp
├── ElectricCar.cpp
├── ElectricCar.hpp
├── ElectricEngine.cpp
├── ElectricEngine.hpp
├── HybridCar.cpp
├── HybridCar.hpp
├── PetrolCar.cpp
├── PetrolCar.hpp
├── PetrolEngine.cpp
├── PetrolEngine.hpp
├── README.md
├── build
│   ├── Cars
│   ├── Cars-catch
│   ├── Makefile
│   ├── cmake_install.cmake
│   └── libCars-lib.a
├── main.cpp
└── test_catch
    ├── CarsTests.cpp
    └── catch.hpp
```

Wykres dziedziczenia klas w projekcie *Cars*



Problem diamentowy pojawia się w przypadku gdy klasa jest pochodną klas o wspólnej klasie bazowej. W momencie wywołania obiektu *HybridCar* wywołuje się dwókrotnie konstruktor *Car*, tj. konstruktor *Car*, konstruktor *PetrolCar,* konstruktor *Car,* konstruktor *ElectricCar,* konstruktor *HybridCar*.
Obiekt *HybridCar* posiada więc 2 kopie składników klasy *Car*.
Rozwiązaniem jest zamieszczanie słowa kluczowego *virtual* w momencie deklarowania sposobu dziedziczenia dla *PetrolCar* i *ElectricCar:*
Class ElectricCar: public virtual Car
Class PetrolCar: public virtual Car

Plik *CMakeLists.txt*

```
set(SRC_LIST
    Car.hpp
    Car.cpp
    ElectricCar.cpp
    ElectricCar.hpp
    ElectricEngine.cpp
    ElectricEngine.hpp
    HybridCar.cpp
    HybridCar.hpp
    main.cpp
    PetrolCar.cpp
    PetrolCar.hpp
    PetrolEngine.cpp
    PetrolEngine.hpp
)
```

W CMakeLists.txt znajduje się lista wszystkich plików źródłowych do kompilacji.
W momencie gdy dodawana jest klasa do nowego pliku źródłowego trzeba pamiętać o rozszerzeniu SRC_LIST.

Plik *main.cpp*

```
#include "PetrolCar.hpp"
#include "ElectricCar.hpp"
#include "HybridCar.hpp"
#include <iostream>

int main()
{
    Car* car = nullptr;
    PetrolCar opel(new PetrolEngine(120, 1800, 6));
    car = &opel;
    car->accelerate(50);
    return 0;
}
```

W #include należy dołączyć pliki nagłówkowe klas obiektów, które tworzy funkcja main()

Struktura plików *.hpp

```
#pragma once
#include "ElectricEngine.hpp"
#include "Car.hpp"


class ElectricCar : public virtual Car
{
protected:
    ElectricEngine* engine_;
    void charge();
public:
    ElectricCar(ElectricEngine* engine);
    ~ElectricCar();
    void feed() override;
    void setElectricEngine(int, int);
};
```

#pragma once – dyrektywa preprocesora sprawiająca, że dany plik źródłowy jest tylko raz uwzględniony podczas kompilacji

W #include zamieszcza się nagłówki klas bazowych oraz klas, których pola/metody są wykorzystywane w pliku

W plikach *.hpp definiuje się dziedziczenie. W tym przypadku, klasa ElectricCar publicznie dziedziczy po klasie Car

Klasa zawiera jedynie deklaracje konstruktorów, destruktorów, pól (zmiennych) oraz metod bez ich źródeł/ciał

Struktura plików *.cpp

```
#include "ElectricCar.hpp"
#include <iostream>

ElectricCar::ElectricCar(ElectricEngine* engine)
    : engine_(engine)
{
    std::cout << __FUNCTION__ << std::endl;
}

ElectricCar::~ElectricCar()
{
    delete engine_;
    std::cout << __FUNCTION__ << std::endl;
}

void ElectricCar::charge()
{
    std::cout << __FUNCTION__ << std::endl;
}
```

W #include należy dołączyć pliki nagłówkowe do definiowanych w pliku źródłowym konstruktorów/metod/destruktorów

W plikach źródłowych definiuje się sposób działania metod/konstruktorów/destruktorów

Podczas definicji metody w nagłówku należy podać nazwę klasy, w której ta metoda została zadeklarowana, używając operatora zasięgu „::”. W tym przypadku definiowany jest destruktor klasy ElectricCar.